26 | MySQL编译器（二）：编译技术如何帮你提升数据库性能？

2020-07-31 宫文学

编译原理实战课 进入课程 >

讲述：宫文学
时长 20:09 大小 18.46M

你好，我是宫文学。今天这一讲，我们继续来探究 MySQL 编译器。

通过上一讲的学习，你已经了解了 MySQL 编译器是怎么做词法和语法分析的了。那么在做完语法分析以后，MySQL 编译器又继续做了哪些处理，才能成功地执行这个 SQL 语句呢？

所以今天，我就带你来探索一下 MySQL 的实现机制，我会把重点放在 SQL 的语义分析和优化机制上。当你学完以后，你就能真正理解以下这些问题了：

> 高级语言的编译器具有语义分析功能，那么 MySQL 编译器也会做语义分析吗？它有没有引用消解问题？有没有作用域？有没有类型检查？

MySQL 有没有类似高级语言的那种优化功能呢？

好，让我们开始今天的探究吧。不过，在讨论 MySQL 的编译过程之前，我想先带你了解一下 MySQL 会用到的一些重要的数据结构，因为你在解读代码的过程中经常会见到它们。

认识 MySQL 编译器的一些重要的数据结构

第一组数据结构，是下图中的几个重要的类或结构体，包括线程、保存编译上下文信息的 LEX，以及保存编译结果 SELECT_LEX_UNIT 和 SELECT_LEX。

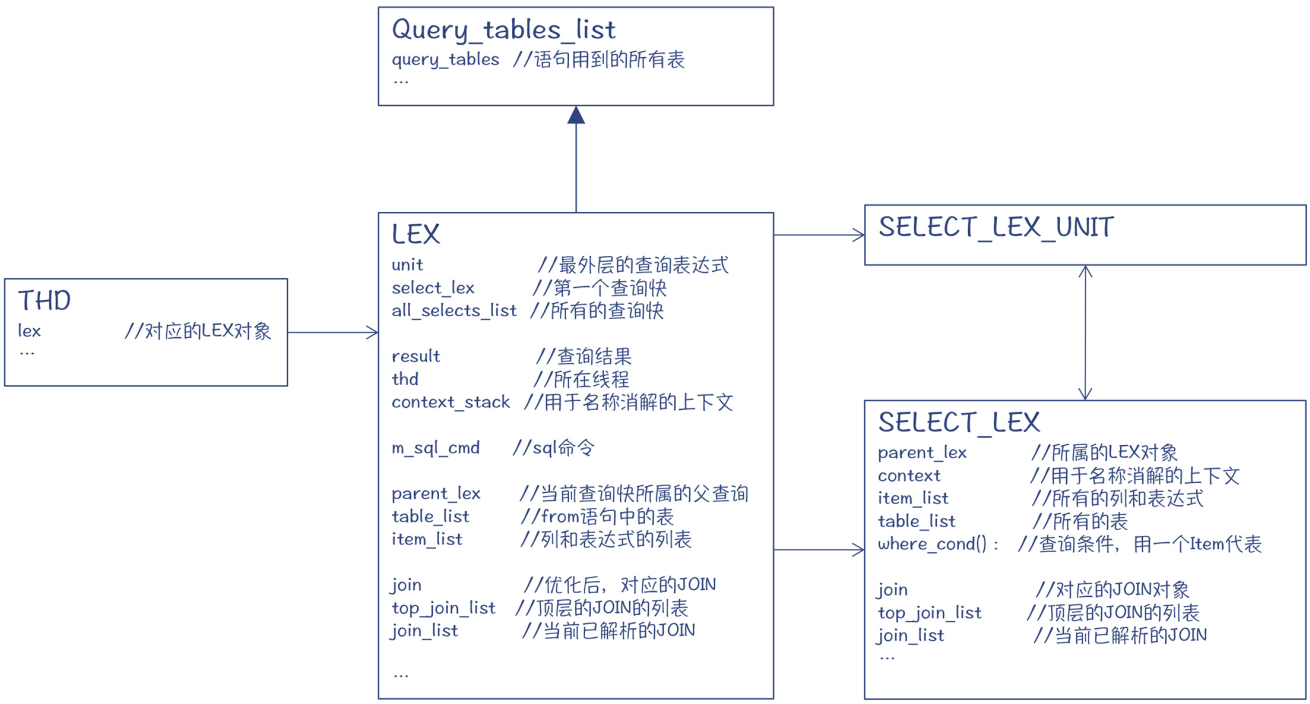


图 1：MySQL 编译器的中的几个重要的类和结构体

首先是 THD，也就是线程对象。对于每一个客户端的连接，MySQL 编译器都会启动一个线程来处理它的查询请求。

THD 中的一个重要数据成员是 **LEX 对象**。你可以把 LEX 对象想象成是编译 SQL 语句的工作区，保存了 SQL 语句编译过程中的上下文信息，编译器会把编译的成果放在这里，而编译过程中所需要的信息也是从这里查找。

在把 SQL 语句解析完毕以后，编译器会形成一些结构化的对象来表示一个查询。其中 **SELECT_LEX_UNIT 结构体**，就代表了一个**查询表达式**（Query Expression）。一个查询

表达式可能包含了多个查询块，比如使用 UNION 的情况。

而 **SELECT_LEX** 则代表一个**基本的查询块**（Query Block），它里面的信息包括了所有的列和表达式、查询用到的表、where 条件等。在 SELECT_LEX 中会保存查询块中涉及的表、字段和表达式等，它们也都有对应的数据结构。

第二组需要了解的数据结构，是表示表、字段等信息的对象。**Table_ident 对象**保存了表的信息，包括数据库名、表名和所在的查询语句（SELECT_LEX_UNIT 对象）。

图 2：Table_indent 对象，代表一个表

而字段和表达式等表示一个值的对象，用 Item 及其子类来表示。SQL 语句中的每个字段、每个计算字段，最后都对应一个 Item。where 条件，其实也是用一个 Item 就能表示。具体包括：

字段（Item_field）。

各种常数，包括数字、字符和 null 等（Item_basic_constant）。

能够产生出值的运算（Item_result_field），包括算术表达式（Item_num_op）、存储过程（Item_func_sp）、子查询（Item_subselect）等。

在语法分析过程中产生的 Item（Parse_tree_item）。它们是一些占位符，因为在语法分析阶段，不容易一下子创建出真正的 Item，这些 Parse_tree_item 需要在上下文分析阶段，被替换成真正的 Item。

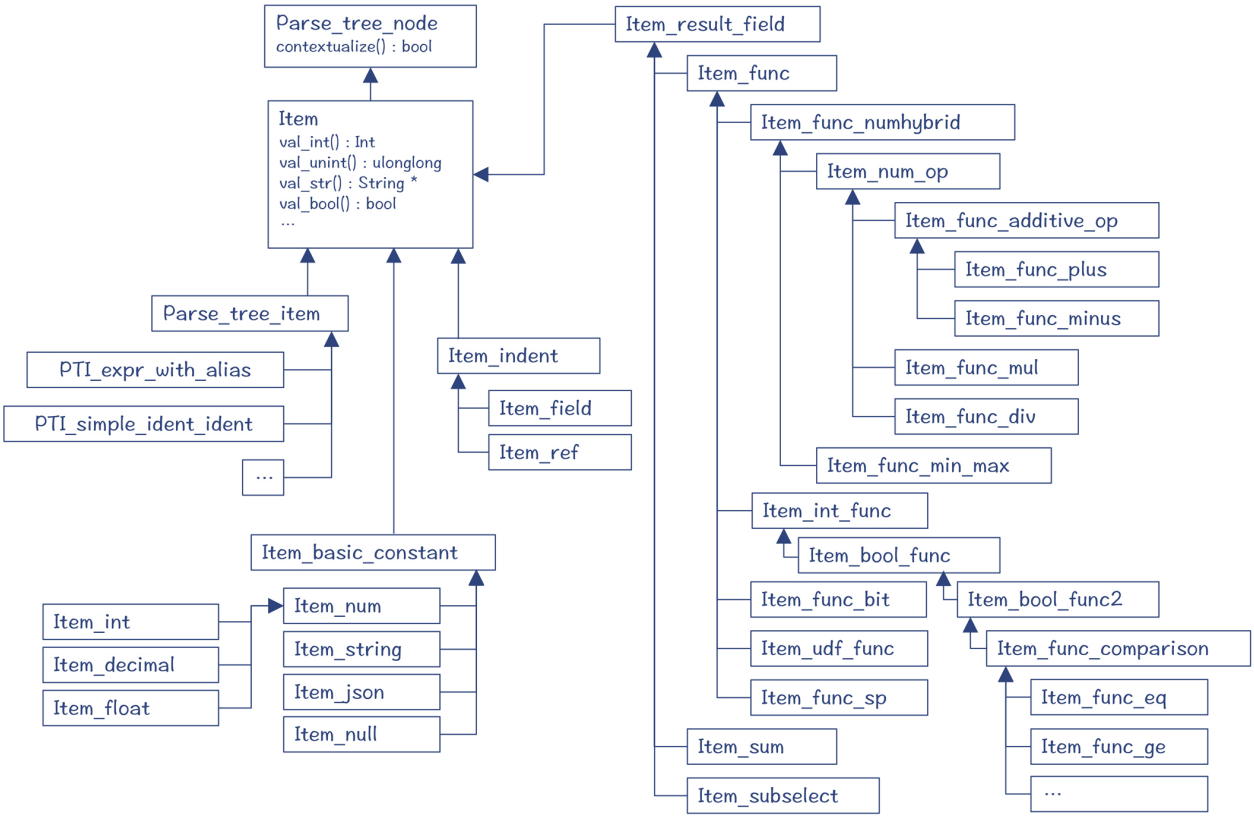


图 3：Item 及其子类

好了，上面这些就是 MySQL 会用到的最核心的一些数据结构了。接下来的编译工作，就会生成和处理上述的数据结构。

上下文分析

我们先来看一下 MySQL 编译器的上下文分析工作。

你已经知道，语法分析仅仅完成的是上下文无关的分析，还有很多的工作，需要基于上下文来做处理。这些工作，就属于语义分析。

MySQL 编译器中，每个 AST 节点，都会有一个 **contextualize() 方法**。从这个方法的名称来看，你就能知道它是做上下文处理的（contextualize，置于上下文中）。

对一个 Select 语句来说，编译器会调用其根节点 PT_select_stmt 的 contextualize() 方法，从而深度遍历整个 AST，并调用每个节点的 contextualize() 方法。

那么，MySQL 编译器的上下文处理，都完成了什么工作呢？

首先，是检查数据库名、表名和字段名是否符合格式要求（在 table.cc 中实现）。

比如，MySQL 会规定表名、字段名等名称不能超过 64 个字符，字段名不能包含 ASCII 值为 255 的字符，等等。这些规则在词法分析阶段是不检查的，要留在语义分析阶段检查。

然后，创建并填充 SELECT_LEX_UNIT 和 SELECT_LEX 对象。

前面我提到了，SELECT_LEX_UNIT 和 SELECT_LEX 中，保存了查询表达式和查询块所需的所有信息，依据这些信息，MySQL 就可以执行实际的数据库查询操作。

那么，在 contextualize 的过程中，编译器就会生成上述对象，并填充它们的成员信息。

比如，对于查询中用到的表，在语法分析阶段就会生成 Table_ident 对象。但其中的数据库名称可能是缺失的，那么在上下文的分析处理当中，就会被编译器设置成当前连接所采用的默认数据库。这个信息可以从线程对象（THD）中获得，因为每个线程对应了一个数据库连接，而每个数据库连接是针对一个具体的数据库的。

好了，经过上下文分析的编译阶段以后，我们就拥有了可以执行查询的 SELECT_LEX_UNIT 和 SELECT_LEX 对象。可是，你可能会注意到一个问题：**为什么在语义分析阶段，MySQL 没有做引用的消解呢？**不要着急，接下来我就给你揭晓这个答案。

MySQL 是如何做引用消解的？

我们在 SQL 语句中，会用到数据库名、表名、列名、表的别名、列的别名等信息，编译器肯定也需要检查它们是不是正确的。这就是引用消解（或名称消解）的过程。一般编译器是在语义分析阶段来做这项工作的，而 MySQL 是在执行 SQL 命令的时候才做引用消解。

引用消解的入口是在 SQL 命令的的 prepare() 方法中，它会去检查表名、列名都对不对。

通过 GDB 调试工具，我们可以跟踪编译器做引用消解的过程。你可以在 my_message_sql() 函数处设个断点，然后写个 SQL 语句，故意使用错误的表名或者列名，来看看 MySQL 是在什么地方检查出这些错误的。

比如说，你可以执行“select * from fake_table”，其中的 fake_table 这个表，在数据库中其实并不存在。

下面是打印出的调用栈。你会注意到，MySQL 在准备执行 SQL 语句的过程中，会试图去打开 fake_table 表，这个时候编译器就会发现这个表不存在。

```
(gdb) bt
#0  my_message_sql (error=1146, str=0x7fb2440ee780 "Table 'employees.fake_table' doesn't exist", MyFlags=0)
    at /home/richard/mysql-server-8.0/sql/mysqld.cc:3365                                          ← 报错误
#1  0x00000000044e3e8a in my_error (nr=1146, MyFlags=0) at /home/richard/mysql-server-8.0/mysys/my_error.cc:249
#2  0x0000000002f7c240 in get_table_share (thd=0x7fb230006fa0, db=0x7fb230a9a7e0 "employees",
    table_name=0x7fb230a99f10 "fake_table", key=0x7fb230a9a617 "employees", key_length=21, open_view=true,
    open_secondary=false) at /home/richard/mysql-server-8.0/sql/sql_base.cc:755
#3  0x0000000002f7c86e in get_table_share_with_discover (thd=0x7fb230006fa0, table_list=0x7fb230a9a230,
    key=0x7fb230a9a617 "employees", key_length=21, open_secondary=false, error=0x7fb2440ef024)
    at /home/richard/mysql-server-8.0/sql/sql_base.cc:860
#4  0x0000000002f81bc7 in open_table (thd=0x7fb230006fa0, table_list=0x7fb230a9a230, ot_ctx=0x7fb2440ef2f0)
    at /home/richard/mysql-server-8.0/sql/sql_base.cc:3160                                        ← 打开table
#5  0x0000000002f85d53 in open_and_process_table (thd=0x7fb230006fa0, lex=0x7fb230009f50, tables=0x7fb230a9a230,
    counter=0x7fb230009fa8, prelocking_strategy=0x7fb2440ef408, has_prelocking_list=false, ot_ctx=0x7fb2440ef2f0)
    at /home/richard/mysql-server-8.0/sql/sql_base.cc:4993
#6  0x0000000002f871d3 in open_tables (thd=0x7fb230006fa0, start=0x7fb2440ef3b0, counter=0x7fb230009fa8, flags=0,
    prelocking_strategy=0x7fb2440ef408) at /home/richard/mysql-server-8.0/sql/sql_base.cc:5648
#7  0x0000000002f88a72 in open_tables_for_query (thd=0x7fb230006fa0, tables=0x7fb230a9a230, flags=0)
    at /home/richard/mysql-server-8.0/sql/sql_base.cc:6503
#8  0x00000000030c867e in Sql_cmd_dml::prepare (this=0x7fb230a9a800, thd=0x7fb230006fa0)
    at /home/richard/mysql-server-8.0/sql/sql_select.cc:448                                       ← 在prepare中做引用消解
#9  0x00000000030c8ff1 in Sql_cmd_dml::execute (this=0x7fb230a9a800, thd=0x7fb230006fa0)
    at /home/richard/mysql-server-8.0/sql/sql_select.cc:655
#10 0x00000000030675b4 in mysql_execute_command (thd=0x7fb230006fa0, first_level=true)
    at /home/richard/mysql-server-8.0/sql/sql_parse.cc:4478                                       ← 执行SQL语句
#11 0x0000000003069f0d in mysql_parse (thd=0x7fb230006fa0, parser_state=0x7fb2440f0c40)
    at /home/richard/mysql-server-8.0/sql/sql_parse.cc:5288
#12 0x000000000305f31d in dispatch_command (thd=0x7fb230006fa0, com_data=0x7fb2440f1cf0, command=COM_QUERY)
    at /home/richard/mysql-server-8.0/sql/sql_parse.cc:1777
#13 0x000000000305d85c in do_command (thd=0x7fb230006fa0) at /home/richard/mysql-server-8.0/sql/sql_parse.cc:1275
#14 0x0000000000321d597 in handle_connection (arg=0x83dd2b0)
    at /home/richard/mysql-server-8.0/sql/conn_handler/connection_handler_per_thread.cc:302
```

你还可以再试一下“select fake_column from departments”这个语句，也一样会查出，fake_column 并不是 departments 表中的一列。

```
(gdb) bt
#0  my_message_sql (error=1054, str=0x7f323c15c850 "Unknown column 'fake_column' in 'field list'", MyFlags=0)
    at /home/richard/mysql-server-8.0/sql/mysqld.cc:3365                                          ← 报错误
#1  0x00000000044e3e8a in my_error (nr=1054, MyFlags=0) at /home/richard/mysql-server-8.0/mysys/my_error.cc:249
#2  0x0000000002f8be02 in find_field_in_tables (thd=0x7f321c006b40, item=0x7f321ca9a390, first_table=0x7f321ca9a4c8,
    last_table=0x0, ref=0x7f321ca9a188, report_error=REPORT_ALL_ERRORS, want_privilege=1, register_tree_change=true)
    at /home/richard/mysql-server-8.0/sql/sql_base.cc:7783
#3  0x000000000342cdba in Item_field::fix_outer_field (this=0x7f321ca9a390, thd=0x7f321c006b40,
    from_field=0x7f323c15cff0, reference=0x7f321ca9a188) at /home/richard/mysql-server-8.0/sql/item.cc:4974
#4  0x000000000342d934 in Item_field::fix_fields (this=0x7f321ca9a390, thd=0x7f321c006b40, reference=0x7f321ca9a188)
    at /home/richard/mysql-server-8.0/sql/item.cc:5254                                            ← 在prepare中做引用消解
#5  0x0000000002f8e240 in setup_fields (thd=0x7f321c006b40, ref_item_array=..., fields=..., want_privilege=1,
    sum_func_list=0x7f321ca99768, allow_sum_func=true, column_update=false)
    at /home/richard/mysql-server-8.0/sql/sql_base.cc:8741
#6  0x00000000030b3da1 in SELECT_LEX::prepare (this=0x7f321ca995f8, thd=0x7f321c006b40)
    at /home/richard/mysql-server-8.0/sql/sql_resolver.cc:247                                     ← 调用SELECT_LEX的prepare方法
#7  0x00000000030c8e42 in Sql_cmd_select::prepare_inner (this=0x7f321ca9aa98, thd=0x7f321c006b40)
    at /home/richard/mysql-server-8.0/sql/sql_select.cc:587
#8  0x00000000030c8799 in Sql_cmd_dml::prepare (this=0x7f321ca9aa98, thd=0x7f321c006b40)
    at /home/richard/mysql-server-8.0/sql/sql_select.cc:470                                       ← 在prepare中做引用消解
#9  0x00000000030c8ff1 in Sql_cmd_dml::execute (this=0x7f321ca9aa98, thd=0x7f321c006b40)
    at /home/richard/mysql-server-8.0/sql/sql_select.cc:655
#10 0x00000000030675b4 in mysql_execute_command (thd=0x7f321c006b40, first_level=true)
    at /home/richard/mysql-server-8.0/sql/sql_parse.cc:4478                                       ← 执行SQL语句
#11 0x0000000003069f0d in mysql_parse (thd=0x7f321c006b40, parser_state=0x7f323c15ec40)
    at /home/richard/mysql-server-8.0/sql/sql_parse.cc:5288
#12 0x000000000305f31d in dispatch_command (thd=0x7f321c006b40, com_data=0x7f323c15fcf0, command=COM_QUERY)
    at /home/richard/mysql-server-8.0/sql/sql_parse.cc:1777
#13 0x000000000305d85c in do_command (thd=0x7f321c006b40) at /home/richard/mysql-server-8.0/sql/sql_parse.cc:1275
#14 0x0000000000321d597 in handle_connection (arg=0x9188e90)
    at /home/richard/mysql-server-8.0/sql/conn_handler/connection_handler_per_thread.cc:302
#15 0x0000000004c7d622 in pfs_spawn_thread (arg=0x9173940)
```

那么，MySQL 是如何知道哪些表和字段合法，哪些不合法的呢？

原来，它是通过查表的定义，也就是数据库模式信息，或者可以称为数据字典、元数据。MySQL 在一个专门的库中，保存了所有的模式信息，包括库、表、字段、存储过程等定义。

你可以跟高级语言做一下类比。高级语言，比如说 Java 也会定义一些类型，类型中包含了成员变量。那么，MySQL 中的表，就相当于高级语言的类型；而表的字段（或列）就相当于高级语言的类型中的成员变量。所以，在这个方面，MySQL 和高级语言做引用消解的思路其实是一样的。

但是，**高级语言在做引用消解的时候有作用域的概念，那么 MySQL 有没有类似的概念呢？**

有的。举个例子，假设一个 SQL 语句带了子查询，那么子查询中既可以引用本查询块中的表和字段，也可以引用父查询中的表和字段。这个时候就存在了两个作用域，比如下面这个查询语句：

复制代码

```
select dept_name from departments where dept_no in
    (select dept_no from dept_emp
        where dept_name != 'Sales'  #引用了上一级作用域中的字段
        group by dept_no
        having count(*)> 20000)
```

其中的 dept_name 字段是 dept_emp 表中所没有的，它其实是上一级作用域中 departments 表中的字段。

提示：这个 SQL 当然写得很不优化，只是用来表现作用域的概念。

好。既然要用到作用域，那么 MySQL 的作用域是怎么表示的呢？

这就要用到 **Name_resolution_context 对象**。这个对象保存了当前作用域中的表，编译器可以在这些表里查找字段；它还保存了**对外层上下文的引用**（outer_context），这样 MySQL 就可以查找上一级作用域中的表和字段。

```
Name_resolution_context

outer_context    //外层的上下文
next_context     //以同一个查询快为基础的下一个上下文
table_list        //用于做名称解析的表
first_name_resolution_table   //第一个表
last_name_resolution_table   //最后一个表

select_lex        //所属的SELECT_LEX
```

图 4：MySQL 用来表示作用域的对象

好了，现在你就对 MySQL 如何做引用消解非常了解了。

我们知道，对于高级语言的编译器来说，接下来它还会做一些优化工作。**那么，MySQL 是如何做优化的呢？它跟高级语言编译器的优化工作相比，又有什么区别呢？**

MySQL 编译器的优化功能

MySQL 编译器的优化功能主要都在 sql_optimizer.cc 中。就像高级语言一样，MySQL 编译器会支持一些常见的优化。我来举几个例子。

第一个例子是常数传播优化（const propagation）。假设有一个表 foo，包含了 x 和 y 两列，那么 SQL 语句：“select * from foo where x = 12 and y=x”，会被优化成“select * from foo where x = 12 and y = 12”。你可以在 propagate_cond_constants() 函数上加个断点，查看常数传播优化是如何实现的。

第二个例子是死代码消除。比如，对于 SQL 语句：“select * from foo where x=2 and y=3 and x<y”，编译器会把它优化为“select * from foo where x=2 and y=3”，把“x<y”去掉了，这是因为 x 肯定是小于 y 的。该功能的实现是在 remove_eq_conds() 中。

第三个例子是常数折叠。这个优化工作我们应该很熟悉了，主要是对各种条件表达式做折叠，从而降低计算量。其实现是在 sql_const_folding.cc 中。

你要注意的是，上述的优化主要是针对条件表达式。因为 MySQL 在执行过程中，对于每一行数据，可能都需要执行一遍条件表达式，所以上述优化的效果会被放大很多倍，这就好比针对循环体的优化，是一个道理。

不过，**MySQL 还有一种特殊的优化，是对查询计划的优化。**比如说，我们要连接 employees、dept_emp 和 departments 三张表做查询，数据库会怎么做呢？

最笨的办法，是针对第一张表的每条记录，依次扫描第二张表和第三张表的所有记录。这样的话，需要扫描多少行记录呢？是三张表的记录数相乘。基于我们的示例数据库的情况，这个数字是 **8954 亿。**

上述计算其实是做了一个**笛卡尔积**，这会导致处理量的迅速上升。而在数据库系统中，显然不需要用这么笨的方法。

你可以用 explain 语句，让 MySQL 输出执行计划，下面我们来看看 MySQL 具体是怎么做的：

复制代码

```
explain select employees.emp_no, first_name,
                departments.dept_no dept_name
        from employees, dept_emp, departments
        where employees.emp_no = dept_emp.emp_no
              and dept_emp.dept_no = departments.dept_no;
```

这是 MySQL 输出的执行计划：

```
mysql> explain select employees.emp_no, first_name, departments.dept_no dept_name
    ->            from employees, dept_emp, departments
    ->            where employees.emp_no = dept_emp.emp_no and dept_emp.dept_no = departments.dept_no;
+----+-------------+-------------+------------+--------+----------------+-----------+---------+-------------------------------+-------+----------+-------------+
| id | select_type | table       | partitions | type   | possible_keys  | key       | key_len | ref                           | rows  | filtered | Extra       |
+----+-------------+-------------+------------+--------+----------------+-----------+---------+-------------------------------+-------+----------+-------------+
|  1 | SIMPLE      | departments | NULL       | index  | PRIMARY        | dept_name | 162     | NULL                          |     9 |   100.00 | Using index |
|  1 | SIMPLE      | dept_emp    | NULL       | ref    | PRIMARY,dept_no | dept_no   | 16      | employees.departments.dept_no | 37450 |   100.00 | Using index |
|  1 | SIMPLE      | employees   | NULL       | eq_ref | PRIMARY        | PRIMARY   | 4       | employees.dept_emp.emp_no     |     1 |   100.00 | NULL        |
+----+-------------+-------------+------------+--------+----------------+-----------+---------+-------------------------------+-------+----------+-------------+
3 rows in set, 1 warning (0.00 sec)
```

从输出的执行计划里，你能看出，MySQL 实际的执行步骤分为了 3 步：

第 1 步，通过索引，遍历 departments 表；

第 2 步，通过引用关系（ref），找到 dept_emp 表中，dept_no 跟第 1 步的 dept_no 相等的记录，平均每个部门在 dept_emp 表中能查到 3.7 万行记录；

第 3 步，基于第 2 步的结果，通过等值引用（eq_ref）关系，在 employees 表中找到相应的记录，每次找到的记录只有 1 行。这个查找可以通过 employees 表的主键进行。

根据这个执行计划来完成上述的操作，编译器只需要处理大约 63 万行的数据。因为通过索引查数据相比直接扫描表来说，处理每条记录花费的时间比较长，所以我们假设前者花费的时间是后者的 3 倍，那么就相当于扫描了 63*3=189 万行表数据，这仍然仅仅相当于做笛卡尔积的 47 万分之一。**我在一台虚拟机上运行该 SQL 花费的时间是 5 秒，而如果使用未经优化的方法，则需要花费 27 天！**

通过上面的例子，你就能直观地理解做查询优化的好处了。MySQL 会通过一个 JOIN 对象，来为一个查询块（SELECT_LEX）做查询优化，你可以阅读 JOIN 的方法，来看看查询优化的具体实现。关于查询优化的具体算法，你需要去学习一下数据库的相关课程，我在本讲末尾也推荐了一点参考资料，所以我这里就不展开了。

从编译原理的角度来看，我们可以把查询计划中的每一步，看做是一条指令。MySQL 的引擎，就相当于能够执行这些指令的一台虚拟机。

如果再做进一步了解，你就会发现，**MySQL 的执行引擎和存储引擎是分开的**。存储引擎提供了一些基础的方法（比如通过索引，或者扫描表）来获取表数据，而做连接、计算等功能，是在 MySQL 的执行引擎中完成的。

好了，现在你就已经大致知道了，一条 SQL 语句从解析到执行的完整过程。但我们知道，普通的高级语言在做完优化后，生成机器码，这样性能更高。那么，**是否可以把 SQL 语句编译成机器码，从而获得更高的性能呢？**

能否把 SQL 语句编译成机器码？

MySQL 编译器在执行 SQL 语句的过程中，除了查找数据、做表间连接等数据密集型的操作以外，其实还有一些地方是需要计算的。比如：

where 条件：对每一行扫描到的数据都需要执行一次。

计算列：有的列是需要计算出来的。

聚合函数：像 sum、max、min 等函数，也是要对每一行数据做一次计算。

在研究 MySQL 的过程中，你会发现上述计算都是解释执行的。MySQL 做解释执行的方式，基本上就是深度遍历 AST。比如，你可以对代表 where 条件的 Item 求值，它会去调用它的下级节点做递归的计算。这种计算过程和其他解释执行的语言差不多，都是要在运

行时判断数据的类型，进行必要的类型转换，最后执行特定的运算。因为很多的判断都要在运行时去做，所以程序运行的性能比较低。

另外，由于 MySQL 采用的是解释执行机制，所以它在语义分析阶段，其实也没有做类型检查。在编译过程中，不同类型的数据在运算的时候，会自动进行类型转换。比如，执行“`select'2' + 3`”，MySQL 会输出 5，这里就把字符串`'2'`转换成了整数。

那么，我们能否把这些计算功能编译成本地代码呢？

因为我们在编译期就知道每个字段的数据类型了，所以编译器其实是可以根据这些类型信息，生成优化的代码，从而提升 SQL 的执行效率。

这种思路理论上是可行的。不过，目前我还没有看到 MySQL 在这方面的工作，而是发现了另一个数据库系统 PostgreSQL，做了这方面的优化。

PostgreSQL 的团队发现，如果解释执行下面的语句，表达式计算所用的时间，占到了处理一行记录所需时间的 56%。而基于 LLVM 实现 JIT 以后（编译成机器码执行），所用的时间只占到总执行时间的 6%，这就使得 SQL 执行的整体性能整整提高了一倍。

```
1 select count(*) from table_name where (x + y) > 100
```

复制代码

中国用户对 MySQL 的用量这么大，如果能做到上述的优化，那么仅仅因此而减少的碳排放，就是一个很大的成绩！所以，你如果有兴趣的话，其实可以在这方面尝试一下！

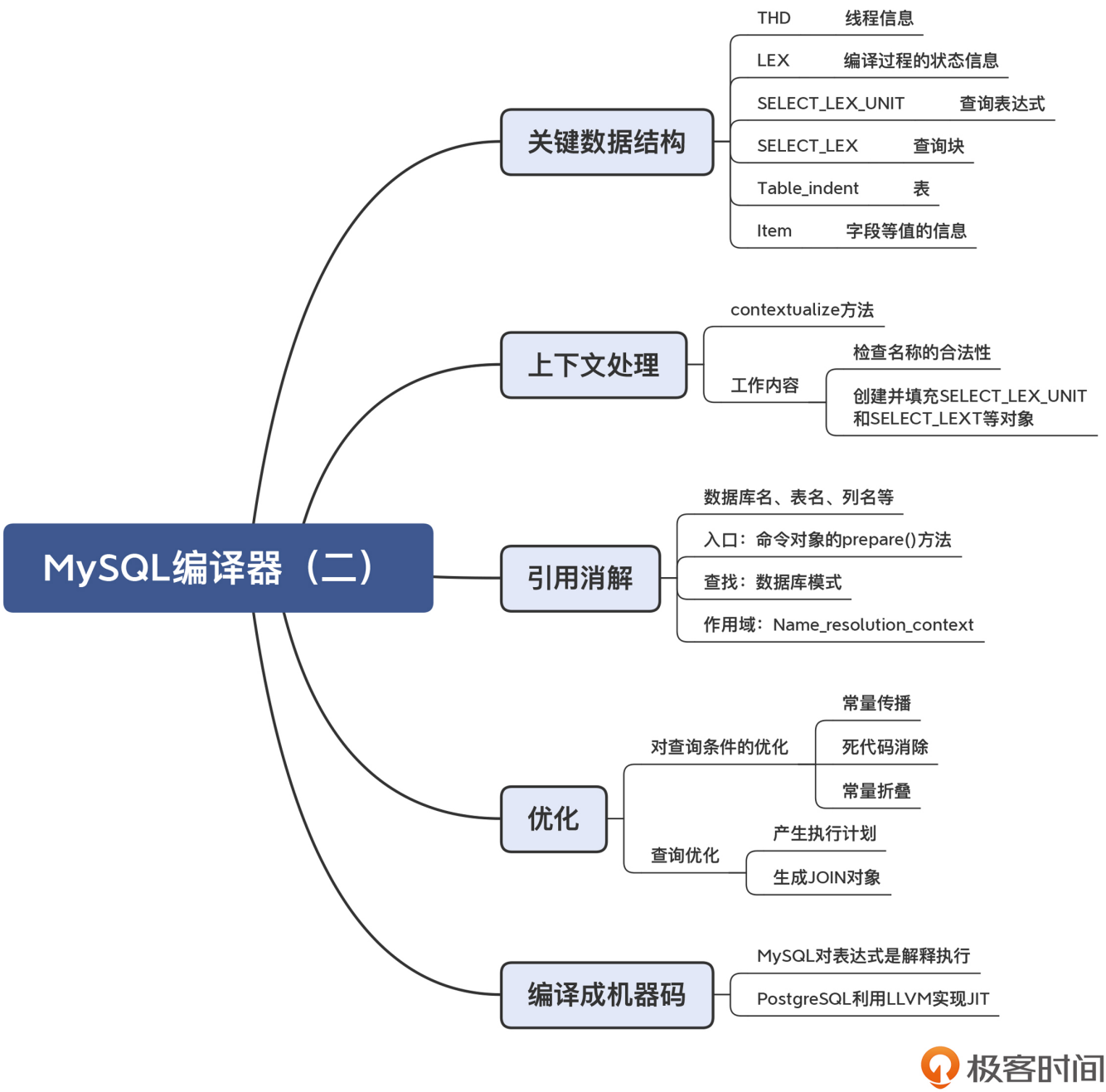
课程小结

这一讲我们分析了 MySQL 做语义分析、优化和执行的原理，并探讨了一下能否把 SQL 编译成本地代码的问题。你要记住以下这些要点：

- MySQL 也会做上下文分析，并生成能够代表 SQL 语句的内部数据结构；
- MySQL 做引用消解要基于数据库模式信息，并且也支持作用域；
- MySQL 会采用常数传播等优化方法，来优化查询条件，并且要通过查询优化算法，形成高效的执行计划；

把 SQL 语句编译成机器码，会进一步提升数据库的性能，并降低能耗。

我把相应的知识点总结成了思维导图，供你参考：



总结这两讲对 MySQL 所采用的编译技术介绍，你会发现这样几个很有意思的地方：

第一，编译技术确实在数据库系统中扮演了很重要的作用。

第二，数据库编译 SQL 语句的过程与高级语言有很大的相似性，它们都包含了词法分析、语法分析、语义分析和优化等处理。你对编译技术的了解，能够指导你更快地看懂 MySQL 的运行机制。另外，如果你也要设计类似的系统级软件，这就是一个很好的借鉴。

一课一思

关系数据库是通过内置的 DSL 编译器来驱动运行的软件。那么，你还知道哪些软件是采用这样的机制来运行的？你如果去实现这样的软件，能从 MySQL 的实现思路里借鉴哪些思路？欢迎在留言区分享你的观点。

参考资料

如果要加深对 MySQL 内部机制的了解，我推荐两本书：一本是 OReilly 的《Understanding MySQL Internals》，第二本是《Expert MySQL》。

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇
下一篇 热点问题答疑 | 如何吃透7种真实的编译器？

精选留言 (2)

写留言

sugar
2020-08-03

浏览器中的html+css 作为ui界面样式，也是一种实用很广泛的dsl。特别是css3 、或一些预处理css工具所衍生出的css方言都会带有表达式计算 甚至自定义函数的能力。

1

myrfy
2020-07-31

正则表达式，sed、awk等文本编辑用的命令行工具。

展开

1